Universitat de Lleida

TREBALL FINAL DE MÀSTER

ESCOLA
POLITÈCNICA SUPERIOR
UNIVERSITAT DE LLEIDA
INSPIRING THE FUTURE

Estudiant: **Albert Eduard Merino Pulido**

Titulació: Màster en Enginyeria Informàtica

Títol de Treball Final de Màster: Automatically Configuring Deep Q-Learning agents for the Berkeley Pacman project

Director/a: Ansótegui Gil, Carlos José

Presentació

Mes: Setembre

Any: 2018

Automatically Configuring Deep Q-Learning agents for the Berkeley Pacman project

Author: Albert Eduard Merino Pulido

Director: Carlos José Ansótegui Gil

Logic and Optimization Group (LOG)
University of Lleida
Lleida, Spain
Friday, September 7th 2018

Acknowledgements

I would like to express my deep gratitude to Carlos Ansótegui, my research supervisor, for his patient guidance, enthusiastic encouragement and useful critiques of this research work. I would also like to thank all the stuff in the Brighton ISE School for their advice and assistance in keeping the English grammar of the project as well as possible. My grateful thanks are also extended to the Heath family who helped me a lot giving me useful English sentences/words to include in the report.

I would also like to extend my thanks to Josep Pont and Jesús Ojeda for their help in offering me resources and help to run the experiments.

Finally, I wish to thank my parents and my girlfriend for their support and encouragement throughout my project.

Abstract

Recently, there have been several advances on integrating Deep Neural Networks (DNNs) and Reinforcement Learning (RL) algorithms. These efforts led to the development of Deep Q-Learning (DQL) algorithms which have been applied successfully to develop competitive approaches for multiagent games. Both DNNs and RL algorithms are highly parameterized and different settings can have a dramatic impact on their efficiency. Thus, DQL algorithms can also greatly benefit from a good setting of their parameters. In this project, we show how to apply Automatic Configuration (AC) tools in order to explore efficiently the parameter search space. We have conducted an extensive experimental investigation in the Berkeley Pacman environment which confirms that AC tools can provide up to an additional 20% boost in performance to DQL agents.

Contents

List of Figures

1 Introduction

One can argue that the ultimate goal of Machine Learning (ML) is to make a machine system that automatically builds models from data without requiring tedious and time-consuming human involvement.

In most cases there is no single ML algorithm (eg. decision trees, random forests, clustering techniques, neural networks, etc.) that provides optimal performance across a broad range of heterogeneous problems. Therefore, modern ML algorithms very frequently expose multiple parameters to users to allow for greater flexibility. Parameters settings fundamentally affect algorithm behaviour and performance, often providing improvements up to 2--5% in challenging benchmarks. These parameters, known as hyper-parameters (to distinguish from standard model parameters), can specify the type of *optimisation algorithm* that is used to train a neural network (stochastic gradient descent, Adam, RMSProp, Adagrad, etc.), the *learning rate* that tells the optimiser how far to move the weights in the direction of the gradient, *regularisation terms* (weight decay, dropout probabilities, etc.), among others. It is widely admitted that finding suitable hyper-parameters (configurations) is difficult and time consuming because the relationship between hyper-parameter settings and algorithm performance is non-trivial and can be very hard to learn, especially when each evaluation of the algorithm being configured is computationally expensive. Setting hyper-parameters automatically reduces manual efforts and can result in orders of magnitude improvements in performance. Thus, we need to apply Automatic Configuration (AC) algorithms to efficiently explore the space of possible configurations. Automated algorithm configuration is emerging as an important technology for the development of novel high-performance ML systems. As an example of this growing trend, Coursera just started offering the course *Improving Deep Neural Networks: Hyper-parameter tuning, Regularisation and Optimisation* [33].

During this project, we will automatically configure the available hyper-parameters of a Deep Q-Learning (DQL) algorithm. In particular, we will work on an extension of the Berkeley Pacman project [24] that implements a DQL agent [40].

To perform the automatic configuration, we will make use of the AC tools developed in the field of Constraint Programming [36]. Over the past decade, a number of methods have been developed in this field for tuning parameters automatically, such as CALIBRA [1], ParamILS [20], I/F-Race [5], SMAC [19], ReACT [8], and GGA [3]. The latter is a gender-based genetic algorithm, developed at the LOG research group in the University of Lleida, we will use an extended version to automatically configure DNNs. This version allows to support transfer learning techniques from parents to children in the evolution process, e.g., the weights of the neural network trained fro the parent.

To the best of our knowledge, the application of these AC techniques to optimize DQLs is a pioneering and promising research avenue.

1.1 Objectives

The aim of this project is to show that Automatic Configuration can be efficiently and effectively applied to Deep Q-Learning applications. To rich this goal is necessary to complete these other objectives:

- Adapt the Berkeley Pacman project to be automatically configured.
- Evaluate the AC tool DGGA on a Deep Q-Learning variation of the Berkeley Pacman project
- Conduct and extensive experimental investigation showing the effectiveness of the approach.

1.2 Document structure

This document is divided in 5 different sections:

- Section 1 corresponds to the introduction of this project.
- Section 2 provides the background on different concepts used in this project and an overview of other similar projects.
- Section 3 describe the architecture of the main classes developed in the project, the implementation of the most important scripts and how to apply DGGA.
- Section 4 discuses all the experiments performed during the project.
- Section 5 presents the conclusion and future work.

2 Background

This section describes concepts used in the project to get a better understanding of these terms. Firstly is described the Pacman Berkeley project, which get all the responsibility of the logic of the Pacman game and the graphics. Then is described the research performed in[40] to get a started project using Deep Q-Learning using this environment. Subsequently, we refresh some general concepts about Reinforcement Learning, Markov Decision Process, Q-learning, Deep Q-Learning and Convolutional Neural Networks. We finalise by introducing Automatic configuration which is the main objective of this project.

2.1 The Berkeley Pacman Project

The Pac-Man projects were developed for UC Berkeley's introductory artificial intelligence course, CS 188. They apply an array of AI techniques to playing Pac-Man. However, these projects don't focus on building AI for video games. Instead, they teach foundational AI concepts, such as informed state-space search, probabilistic inference, and reinforcement learning. These concepts underly real-world application areas such as natural language processing, computer vision, and robotics.

They designed these projects with three goals in mind. The projects allow students to visualise the results of the techniques they implement. They also contain code examples and clear directions, but do not force students to wade through undue amounts of scaffolding. Finally, Pac-Man provides a challenging problem environment that demands creative solutions; real-world AI problems are challenging, and Pac-Man is too.

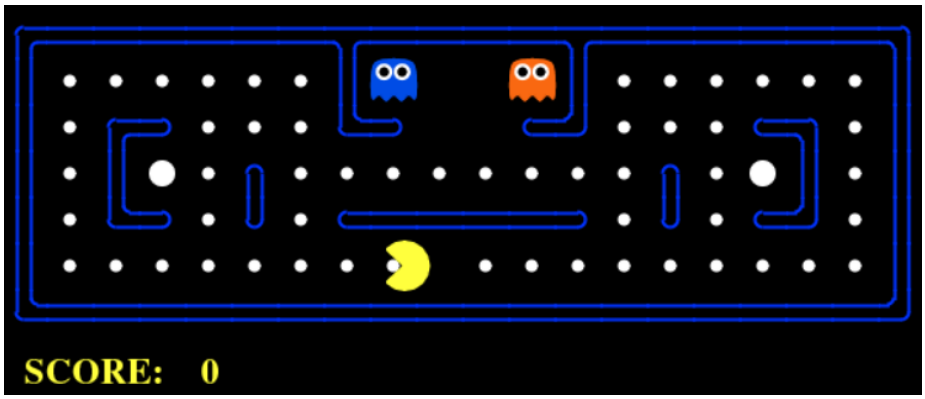

Figure 1: Example of map in the Berkeley Pacman environment

2.2 Machine Learning (ML)

Machine learning, a sub-field of Artificial Intelligence, explores the study and construction of algorithms that can learn from and make predictions on data. ML is employed in a range of computing tasks where designing and programming explicit algorithms with good performance is difficult or unfeasible. Example applications include detection of network intruders or malicious insiders working towards a data breach, email filtering, optical character recognition, shop recommendations, and computer vision or machine listening.

Within ML we have 3 main families of algorithms:

- **Supervised:** the data consists of a set of items, where each item is described by a set of attributes. There is an special attribute called the target or class attribute which indicates the class the item belongs to. The goal is to be able to predict the value of the class attribute for items where it is missing. The machine learning process consists of two phases: the training and the test phase.

  During the training phase, the ML algorithm builds a *predictor* (or classifier) based on a dataset where the target or class attribute is known for every item. These items are referenced as examples, or labelled data. During the test phase, the predictor is used to classify unlabelled data.

- **Unsupervised:** In this case there are no examples classes or targets; all the data is unlabelled. The goal of the ML algorithm can be then, for instance, to discover the possible classes of items by clustering those that are more similar in terms of a similarity function applied to the attributes of the items. Another possibility could be to learn the underlying distribution of the data in order to generate credible new instances of the data. This has clear applications for data augmentation and synthesis. In addition, forecasting problems could be assumed to belong to the unsupervised category, as after the recollection of past sequential data the next point to be predicted becomes the input instance for a further prediction step.

- **Reinforcement Learning:** In this form of learning, we work with a set of states and a set of actions. The actions allow to move from one state to another. The goal is to learn which sequence of actions to take, to move from any of the initial states to a final state that maximises some objective function or score.

There are several frameworks for supervised and unsupervised machine learning: Microsoft Azure Machine Learning, BigML, scikit-learn, Weka, etc. There are also specific frameworks for DNNs which can be used in supervised and unsupervised scenarios:

- `Caffe`: Caffe is a deep learning framework made with expression, speed, and modularity in mind. Caffe is developed by the Berkeley Vision and Learning Center (BVLC), as well as community contributors and is popular for computer vision.

- `Caffe2`: Built on the original Caffe, Caffe2 is designed with expression, speed, and modularity in mind, allowing for a more flexible way to organize computation.
- `Microsoft Cognitive Toolkit`: The Microsoft Cognitive Toolkit is a unified deep-learning toolkit from Microsoft Research that makes it easy to train and combine popular model types across multiple GPUs and servers.
- `TensorFlow`: TensorFlow is a software library for numerical computation using data flow graphs, developed by Google's Machine Intelligence research organization.
- `Theano`: Theano is a math expression compiler that efficiently defines, optimizes, and evaluates mathematical expressions involving multi-dimensional arrays.
- `Torch`: Torch is a scientific computing framework that offers wide support for machine learning algorithms.
- `MXnet`: MXnet is a deep learning framework designed for both efficiency and flexibility.
- `Chainer`: Chainer is a deep learning framework that's designed on the principle of define-by-run. Unlike other frameworks, Chainer lets you modify networks during runtime, allowing you to use arbitrary control flow statements.
- `Keras`: Keras is a minimalist, highly modular neural networks library, written in Python, and capable of running on top of either TensorFlow or Theano. Keras was developed with a focus on enabling fast experimentation.

In this project we will use *Tensorflow* since it allows low level access to all the details of the Neural Networks. Tensorflow is a python package that provides two high-level features: Tensor computation with strong GPU acceleration and Deep Neural Networks built on a tape-based autograd system.

2.3 Reinforcement Learning and Markov Decision Process

Reinforcement Learning (RL) is an area of Machine Learning where an agent interacts with an environment to learn what actions it needs to take in any given environment state to maximise the reward. The mathematical framework for defining a solution in reinforcement learning scenario is called Markov Decision Process (MDP). This can be designed as a set of states S, a set of actions A, a reward function R, a policy π and value V. The next diagram shows how it works a RL agent in general together with MDP.

The agent have to take an action (A) to transition from our start state to our end state (S). In return getting rewards (R) and an observation for each action

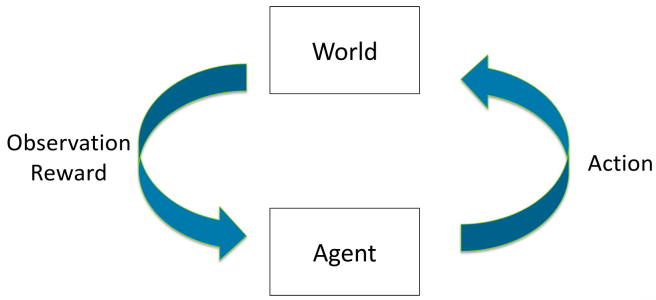

Figure 2: General Reinforcement Learning diagram

we take from the world. Our actions can lead to a positive reward or negative reward. The observation give all the information about the current state.

The set of actions we took define our policy (π) and the rewards we get in return defines our value (V). Our task here is to maximise our rewards by choosing the correct policy. So we have to maximise for all possible values of S for a time t.

In reinforcement learning the basic and most simple way to no take the first found solution is the epsilon-greedy strategy. Using this strategy, the agent selects some random action in the ϵ cases (exploration), in the other cases the agent just follows its so far best found path (exploitation), see figure 2.3. This ensures that the agent with some probability explores all possible states and therefore it can find some better path than it found before. This technique is very simple, but it has one big drawback that the exploration, usually random selection, selects from all states uniformly. This means that even the very bad options will be selected quite often so the agent takes longer time to learn the optimal path.

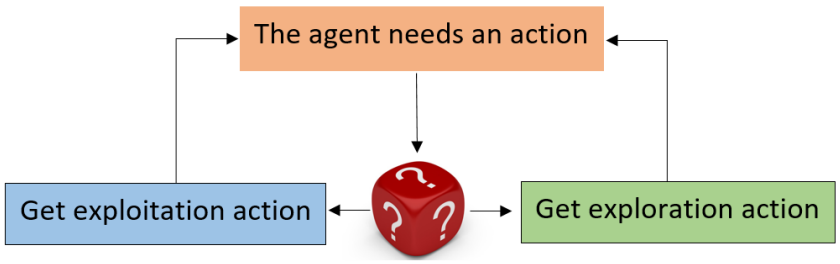

Figure 3: Exploration vs Exploitation

2.4 Neural Networks (NNs)

Neural Networks are inspired in biology, more concretely on the brain and how neurons are connected. The first model represented an artificial neuron, known as the perceptron [30], see Figure 4. The image shows how an output y is generated from the combination of the inputs $\mathbf{x}$ and some weights $\mathbf{w}$ as $y = f(\sum w_i x_i)$. This function f is the activation function of the neuron and acts as a binary classification, it squashes the resulting values.

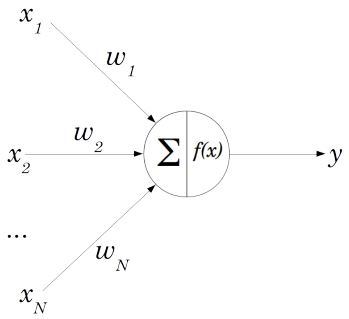

Figure 4: Perceptron.

Assembling several of these neurons, connecting the output of some of them as the inputs of others, we achieve a "multilayer perceptron" (MLP) that can represent a more complex function. This MLP can also be thought of as a network of neurons, a *neural network*. The goal of a neural network is, then, to approximate some complex function f^*. For example, for a classifier, $y = f^*(\mathbf{x})$ maps an input $\mathbf{x}$ to a category y. A feedforward network defines a mapping $\mathbf{y} = f(\mathbf{x}; \theta)$ and learns the value of the parameters $\theta = (w_1 ... w_N)$ that result in the best function approximation. These models are called feedforward because information flows through the function being evaluated from $\mathbf{x}$, through the intermediate computations used to define f, and finally to the output $\mathbf{y}$. There are no feedback connections in which outputs of the model are fed back into itself. When feedforward neural networks are extended to include feedback connections, they are called recurrent neural networks.

Neural networks are called networks because they are typically represented by composing together many different functions. The model is associated with a directed acyclic graph describing how the functions are composed together. For example, we might have three functions $f^{(1)}$, $f^{(2)}$, and $f^{(3)}$ connected in a chain, to form $f(x) = f^{(3)}(f^{(2)}(f^{(1)}(x)))$. These chain structures are the most commonly used structures of neural networks. In this case, $f^{(1)}$ is called the first layer of the network, $f^{(2)}$ is called the second layer, and so on.

2.5 Neural Networks Hyper-parameters

The common hyper-parameters exposed by Tensorflow and generally all deep learning frameworks and toolkits are described below:

- *Learning rate*: $[0, \infty] \in \mathbb{R}$. With the learning rate parameter the optimisation algorithm moves through the gradient landscape of all possible functions that could be approximated by this NN, searching for the lowest point, the minimum, which will be the optimum approximation to the function that best describes the data used for training. This parameter has a very broad range and how well it works depends in great measure in the chosen optimiser.

- *Batch size* : $[1, \infty] \in \mathbb{Z}$. With actual hardware, it is best utilised by using their parallelized pipelines. With regards to NNs, this means how many data instances, e.g. in a supervised training task, can be processed together in a single go, a batch. It is obviously restricted by the amount of memory the computing platform has. The optimiser will perform an update of the model parameters for each batch of examples processed.

- *Optimiser*: Generally speaking, the traditional training method for NN is the classical Gradient Descent algorithm for optimisation. The optimizers that Tensorflow provides are derivations and improvements on the traditional Gradient Descent and we enumerate some of them here with their dependent hyper-parameters (some of them shared among several optimizers):

  - *Stochastic Gradient Descent (SGD)* SGD performs frequent updates with a high variance that cause the objective function to fluctuate heavily. On the one hand, this enables it to jump to new and potentially better local minima. On the other hand, the fluctuation ultimately complicates convergence to the exact minimum, as SGD will keep overshooting.

    * *Momentum* : $[0, \infty) \in \mathbb{R}$, default as 0, recommended in the literature to be within $(0.5, 0.9)$. Helps accelerate SGD in the relevant direction.
    * *Weight decay* : $[0, \infty) \in \mathbb{R}$, default as 0. ℓ_2 penalty to avoid high numbers in the model parameters.
    * *Dampening* : $[0, \infty) \in \mathbb{R}$, default as 0. Dampens oscillations.
    * *Nesterov* : Boolean. Nesterov modifies the momentum computation to direct it to the minima in less steps.

  - *Adagrad* It adapts the learning rate to the parameters, performing larger updates for infrequent and smaller updates for frequent parameters.

    * *Learning rate decay* : $[0, \infty) \in \mathbb{R}$, default as 0. Decay to apply to the learning rate, similar to a learning rate scheduling.
    * *Weight decay* : $[0, \infty) \in \mathbb{R}$, default as 0. ℓ_2 penalty to avoid high numbers in the model parameters.

  - *Adadelta* Adadelta is an extension of Adagrad that seeks to reduce its aggressive, monotonically decreasing learning rate.

        * *Rho (ρ)* : $[0, 1] \in \mathbb{R}$, default as 0.9. Coefficient used for computing a running average of squared gradients.
        * *Eps (ϵ)* : $[0, \infty] \in \mathbb{R}$, default as $1e-6$. Term added to improve numerical stability.
        * *Lr* : $[0, 1] \in \mathbb{R}$, default as 1. Coefficient that scale delta before it is applied to the parameters.
        * *Weight decay* : $[0, \infty) \in \mathbb{R}$, default as 0. ℓ_2 penalty to avoid high numbers in the model parameters.
    - *Rprop* Rprop takes into account only the sign of the partial derivative over all patterns (not the magnitude), and acts independently on each weight.
        * *Etas (η^-, η^+)* : $[0, \infty] \in \mathbb{R}$, default as $(0.5, 1.2)$. Multiplicative increase and decrease factors.
        * *Step Sizes (min, max)* : $[0, \infty] \in \mathbb{R}$, default as $(1e-6, 50)$. A pair of minimal and maximal allowed step sizes.
    - *RMSprop* Similar to Adadelta, RMSprop also divides the learning rate by an exponentially decaying average of squared gradients.
        * *Lambda (λ)* : $[0, 1] \in \mathbb{R}$, default as 0.99. Smoothing constant.
        * *Eps (ϵ)* : $[0, \infty] \in \mathbb{R}$, default as $1e-6$. Term added to improve numerical stability.
        * *Momentum* : $[0, \infty] \in \mathbb{R}$, default as 0. Momentum factor.
        * *Weight decay* : $[0, \infty] \in \mathbb{R}$, default as 0. ℓ_2 penalty to avoid high numbers in the model parameters.
        * *Centered*
    - *Adam* Adaptive Moment Estimation (Adam) is another method that computes adaptive learning rates for each parameter. In addition to storing an exponentially decaying average of past squared gradients like Adadelta and RMSprop, also keeps an exponentially decaying average of past gradients.
        * *Betas (β_1, β_2)* : $[0, 1] \in \mathbb{R}$, default as $(0.9, 0.99)$. Coefficients used for computing running averages of the gradient and its square.
        * *Eps (ϵ)* : $[0, \infty] \in \mathbb{R}$, default as $1e-8$. Term added to improve numerical stability.
        * *Weight decay* : $[0, \infty] \in \mathbb{R}$, default as 0. ℓ_2 penalty to avoid high numbers in the model parameters.

Other parameters that can also be considered relevant as hyper-parameters include:

- *Number of epochs*: $[0, \infty] \in \mathbb{Z}$. A neural network learns by "seeing" data instances from the dataset of the task at hand. An iteration over all the data instances in a dataset is known as an epoch. The training of a neural network is usually done by iterating multiple times over the dataset. Too

many epochs and the NN will overfit to the training data and will have poor results on general data, too few epochs and the NN will not have learnt enough.

- *Learning rate schedule*: As the training of the NN progresses, it is common practice to change dynamically the learning rate to better approach the gradient minimum as to not overshoot it and get more accuracy on the result. How the learning rate changes through training time may be defined with a function or policy, also known as schedule, and it may depend on different factors such as number of epochs already trained, validation accuracy, etc.

- *Early stopping conditions*: When training large models, they may be subject to overfitting, which can be seen with a lowering training error but an increasing validation error. Once this situations happens, it is unlikely that the model will improve and it is common to stop the training. This condition may be implemented through a "patience" parameter that limits the number of times the accuracy on the validation set can be worsened before stopping the training altogether,

- *Architecture related parameters*: When creating a NN, some decisions of how it will work are required. These include the type of computational units used, how many of them are and in which layout, how are the connections made, etc.

  - *Dropout* Dropout is used to add regularisation to the training, reduce overfitting and more specifically prevent co-adaptation of the computational units or neurons. It works by randomising the elements which are zeroed from a NN layer by a certain probability.
  - *Normalisation (batch norm, layer norm, etc.)* Adding normalisation to a layer before being input to the next layer. The primary object is to improve overall optimisation but it may also help with gradient issues.
  - *Number of layers (depth) and their size (width)* How many layers and how many units each layer has can be thought as a parameter on the building of the architecture. They will define the capacity of the net, that is, its ability to fit a wide variety of functions, from which we want to find the best one that defines the data of the problem we train the NN with.
  - *Type of units/blocks, activations and their connections* When creating an architecture, each layer will have some kind of unit, .e.g. in image related tasks it is common to use convolutional layers while for sequence related problems is usual to use a special recurrent unit like the LSTM [16]. In addition to this, one decision that should be made is how each layer is connected to the next one, or even if connects directly to some posterior layer after the next one (creating what is

known as a skip or shortcut connection [14, 17]). Furthermore, in the case of convolutional layers, besides the convolution parameters, another family of operations known as pooling can be added to reduce dimensionality and make the representation become invariant to small translations of the input. Additionally, each unit uses some activation function that adds a non-linear mapping to the computation. Although the default recommendation is the ReLU [23, 11], there are more types of activation functions (sigmod, tanh, etc.) that can be used, each one with its particular behaviour. Finally, in the general case one can define a set of units that may work together and use that set as a building block to create new NNs.

In the case of this project we just use the common parameters.

2.6 Deep Neural Networks (DNNs)

Deep Neural Networks are in its basic definition Neural Networks with several hidden layers, those that are neither input nor output to the network. As more layers are added to a NN, its capability to represent more functions increases. The number of model parameters required for an equivalent shallower neural network increases exponentially, as the shallower net should be much wider.

Besides the typical hyper-parameters, network depth and architecture are key. As an example, the Imagenet challenge [37] has seen a progressive increment on the depth of the competing networks year after year [26, 38, 22, 14]. As the depth increases, a problem due to the gradient descent algorithm and the backpropagation implementation from back to front layers may appear: the vanishing gradient problem. The model parameters are updated following the gradient direction and the loss for a given data instance has to be distributed along the units in the network. As more and more layers of units are built in a given architecture, the loss will become lower and lower to the front layer units. It may even become zero, so the learning would stop. The opposite problem may also occur, high gradients in the optimisation will result in large updates to the model parameters. This will turn the network unstable and the model parameters can become so large as to overflow, which is known as the exploding gradients problem. As a whole DNN architecture example, Figure 5 shows and compares ResNet approach with 34 layers with VGGNet [38] with 19 layers for the ImageNet dataset.

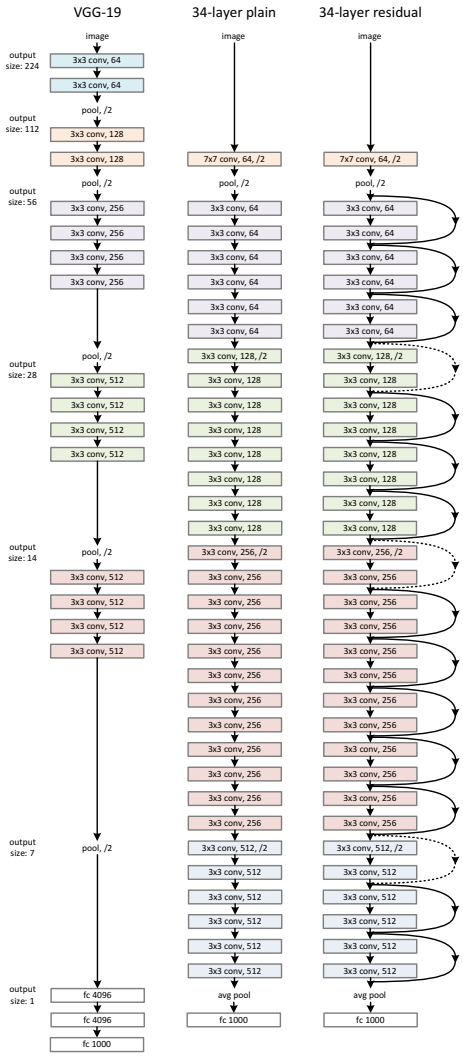

Figure 5: Comparison between VGGNet-19 [38] and ResNet-34 (with and without skip connections) [14] for the ImageNet dataset. Each *conv* block represent a convolutional layer with the respective parameters (kernel size, channels and image size reduction). Each *pool* layer represents an average pooling operation that reduces the size of the image in half. Both models use linear layers of full connected neurons (*fc*) to predict the class of processed image from the dataset (which has a total of 1000 classes). The difference from the plain and the residual models resides just in the use of the identity mappings, represented here by the black skipping arrows in the residual model. Image borrowed from [14].

2.7 Deep Q-Learning

Q-learning is a simple way for agents to learn how to act optimally in controlled Markovian domains. Q-Learning is one of the most popular of value reinforcement learning. Conceptually, Q-Learning algorithms focus on learning a Q-Function that qualifies a state-action pair. A Q-Value represents the expected long term reward of a Q-Learning algorithm assuming that it takes a perfect sequence of actions from a specific state. The Q-function is updated to match the expected discounted future reward using the function using the follow function:

$$Q_\theta(s_t, a_t) \leftarrow Q_\theta(s_t, a_t) + \alpha * (r_{t+1} + \gamma * \max_a Q_\theta(s_{t+1}, a) - Q_\theta(s_t, a_t))$$

where $Q_\theta(s_t, a_t)$ is the Q function, α is a learning rate, s and a are respectively the state and the action, r_{t+1} is the reward corresponding with the current state transition and γ is the discount factor.

Neural networks are the agent that learns to map state-action pairs to rewards. They use coefficients[28] to approximate the function relating inputs to outputs, and their learning consists to find the right coefficients, or weights, by iteratively adjusting those weights along gradients that promise less error.

In reinforcement learning, convolutional networks can be used to recognise an agent's state where usually they perform image recognition. But convolutional networks derive different interpretations from images in reinforcement learning than in supervised learning. In supervised learning, the network applies a label to an image so usually it matches names to pixels.

2.8 Automatic configuration algorithms

Several approaches exist in the literature for the automatic tuning of algorithms. The first methods were created for tuning specific algorithms for a certain task. [31] devised a modular algorithm for solving constraint satisfaction problems (CSPs) and used a combination of exhaustive enumeration of all possible configurations and a parallel hill-climbing technique to automatically configure the system for a given CSP with an associated set of training instances. [10] classified local search (LS) approaches for SAT by means of context-free grammars and devised a genetic programming approach to select a good LS algorithm for a given set of SAT problems. [35] embedded a sequential parameter optimisation approach in a wider framework for the design of evolutionary algorithms.

To tune the continuous parameters of general algorithms, [7] suggested an approach that determines good parameters for individual training instances. These parameters are found by trying configurations where parameters are at their extreme values and then fitting a regression function to the parameter/value tuples obtained in this way. The minimisation of the resulting function yields a set of parameters for the given instance. A parameter set for the entire collection of instances was then obtained by averaging the parameter tuples for the individual instances.

Tuning problems with small sets of parameter configurations were considered

in [6], a setting which is closely related to that in algorithm portfolios [13, 18]. In this case, it is possible to race the different algorithms against each other, whereby a statistical test is used to eliminate inferior algorithms before the remaining algorithms are run on the next training instance.

In [34] Oltean used evolutionary algorithms by means of linear genetic programming. The genome of an individual is an encoding of an actual C-program for the problem to be solved, and crossover and mutation operators are problem dependent. The linear genetic program generates new individuals which replace the current worst individual in the population.

The CALIBRA system, proposed by [2], starts with a factorial design of the parameters. Once these initial parameter sets have been run and evaluated, an intensifying local search routine is started from a promising design, whereby the range of the parameters is limited according to the results of the initial factorial design experiments. The only system we know of that can configure arbitrary algorithms with very large numbers of parameters was proposed by [21]. Their system, called ParamILS, conducts an iterated local search, whereby a special technique is used to limit the number of training instances that need to be run for each parameter set by focusing the test runs on promising parameter sets. In particular, a new set of parameters is not considered better than the current best until it has been evaluated on at least as many training instances as the current best. If a very large set of training instances is available, this approach allows quick movement through the search space while still avoiding an "over-tuning" effect which would be caused by considering few training instances only.

2.8.1 Gender based Genetic Algorithm (*GGA*)

In [4] it is proposed a genetic algorithm for the problem of configuring solvers. There are two main reasons for this choice of approach. First of all, genetic algorithms are known to be very robust with respect to optimisation problems that have undesirable objective landscapes [12]. Note that, in ordinary optimisation, we usually have the freedom to adjust the objective in such a way that it is better suited for sequential local search which often yields good solutions faster than population-based approaches. In contrast, in our setting, where the target algorithm is given and the effect of changing parameters is a priori unknown, we must be able to cope with whatever objective landscape we encounter. The other reason is that genetic algorithms are inherently parallel. When trying to assess which individuals are competitive (the most time-intensive step in solver configuration), genetic algorithms allow us to race them against each other. Therefore, the time spent for the evaluation is determined by the good parameter sets, and this saves a lot of time in practice. In order to really exploit this last aspect, in [4] it is introduced the concept of gender in the genetic algorithm. It is proposed to apply different selection pressure on the two gender populations. In particular, it is applied intra-specific competition only in one part of the population (competitive individuals or genomes). Individuals in this group must compete for the right of mating, and only the fittest in each generation win the right to mate with some of the individuals of the opposite

gender (non-competitive individuals or genomes). The individuals in this other group are not subjected to intra-specific selection.

In the following, we describe the specific *GGA* that follows the structure of a typical Genetic Algorithm.
GGA distinguish three types of target algorithm parameters: continuous and integer parameters, both associated with an upper and lower bound, and categorical values that come with an explicit list of feasible values. In addition *GGA* has the input parameters (*X, P, M, A, S*) which are used in the following way:

- **Initialisation:** first, we randomly initialise the population and assign a gender C (for competitive) or N (for non-competitive) and an "age" of 1 to A years uniformly at random to each individual. In our experiments, we set A to 3.

- **Mating Rules:** among the individuals with gender C, we select the top $X\%$ (in our experiments we set X to 10%). These have gained the right to mate in this season. $200/A\%$ of individuals of gender N are assigned uniformly at random to one of the mating individuals of gender C. The individuals of gender C then mate with all individuals of gender N which have been assigned to them.

- **Crossover:** each mating of a couple of genomes results in one new genome with age 0 and random gender. The genome of the offspring is determined by traversing the parameter tree top-down.

  The parameter tree is the representation of the target algorithm parameters as an And/Or tree (see e.g [29]), where:

  - Each node is labelled with a parameter or the additional label "&", and each algorithm parameter is associated with at least one node.
  - Nodes associated with continuous or integer parameters have at most one child, and And-nodes have at least two child-nodes.
  - The children of categorical nodes partition the set of values that their parent parameter can take. Branches leading to the children are labelled by the respective value(s) of the categorical parameter.

- **Mutation:** as a final step to determine the offspring's genome, with probability $M\%$ we mutate the value of each parameter (in our experiments, we set M to 10%). If we mutate a categorical parameter, we choose a new value in its domain uniformly at random. For continuous and integer parameters, we choose a new value according to a Gaussian distribution where the current value marks the expected value and the variance is set as $S\%$ of the parameter's domain. In our experiments, we set S to 10%.

- **Death:** after the new offspring is created, all individuals' ages are increased by 1. Those with age greater than A are removed from the population. In combination with the mating rules that only $200/A\%$ of individuals of gender N mate in every season, this stabilises the total population size.

2.9 Distributed architectures for high performance computing

A distributed computer system consists of multiple software components that are on multiple computers, but run as a single one. There is no clear distinction between "distributed computing" and "parallel computing", in fact the same system may be characterised as "parallel" and "distributed". A distributed system composed by multi-core computers is inherently running concurrently in parallel.

We can consider that parallel computing is composed by tightly coupled components, while the components of a distributed computing system are loosely coupled. Therefore we can classify concurrent systems using the following criteria:

- In parallel computing, all processors may have access to a shared memory to exchange data.
- In distributed computing, the processors do not have access to shared memory and data is exchanged by passing messages between them.

Current clusters for high performance computing allow to have both types of concurrent systems since as we will see in the next section, they are composed by nodes with their own private memory that can cooperate. Moreover, each node usually corresponds to a multi-core computer where all jobs running on it can access to shared memory.

In this project, since we will follow a Master-Worker pattern where Workers run on independent computers, we will use the term "distributed" to reference the parallelization of *GGA*.

2.9.1 Computer cluster

A computer cluster is a set of loosely or tightly connected independent computers, with common hardware components, that work together as a single system. The computers of a cluster are usually connected to each other through fast Local Area Networks (LAN) and share the same file system.
Within a computer cluster we can identify three different types of nodes:

- **Master node**: it is in charge of the cluster administration and has the scheduler and several parallel libraries.
- **Submit/Interactive nodes**: they are the users entry points and are mainly used to launch jobs.
- **Computational nodes**: these are the nodes where the jobs are executed.

2.9.2 Job scheduler

A job scheduler, also known as distributed resource management system (DRMS), is a computer application for controlling unattended background program execution. It is the responsible of the properly distribution of the resources of a distributed system among the requested jobs.

A global view of a job scheduler has two main modules:

- A **Distributed resource management** (DRM) module, which is in charge of managing the resources of the system.
- A **Scheduler** module, which is the responsible to instruct to the DRM what to do with the available resources. It is also in charge of monitoring the jobs and managing them at run time.

The scheduler module uses different policies to grant that all the jobs can be executed avoiding resource conflicts between them. Some examples are:

- **First In First Out** (FIFO): it is the simplest algorithm. It simply queues jobs in the order that they arrive.
- **Round-Robin** (RR): the scheduler assigns a time unit per job and cycles through them.
- **Shortest Job First** (SJF): it requires an estimation of the required execution time of each jobs. Then, the jobs are arranged based on that estimation.
- **Multilevel queue**: this policy is used when the jobs can be classified into groups based on properties like CPU time or IO access. Each queue has a preassigned priority and its own scheduling algorithm.

There are several implementations of job schedulers, but in this project we are focused in the batch-queue schedulers, which are the most widely used in computer clusters.

A batch-queue scheduler is an specific implementation of a job scheduler. In this type of schedulers the available resources are grouped in one or several queues. By grouping resources, a queue is automatically imposing some resource limitations to the jobs. Besides this physical limitations, a queue can also be configured to force additional logical limitations or requirements to the jobs.

2.9.2.1 SGE The Sun Grid Engine (SGE), originally developed by Sun Microsystems and continued by Oracle under the name Oracle Grid Engine, is a batch-queue system used in high-performance computing (HPC) clusters running on UNIX-like operating systems or Windows with the SFU or SUA extensions. The system is responsible of accepting, scheduling, dispatching and managing, standalone, parallel or interactive user jobs and the resources requested for those jobs.

3 Architecture and Implementation

This section describes all the details about how it is implemented the full project, including the Berkeley Pacman project architecture, the Neural Network Architecture, the Pacman parameters and how to apply DGGA int the project.

3.1 Pacman Architecture

Although the Pacman project had already created, here are described all the files in the project.

- layouts: Directory with all the layouts files
- DQN.py: Implementation of the neural network
- game.py: The logic behind how the Pacman world works. This file describes several supporting types like AgentState, Agent, Direction, and Grid.
- ghostAgents.py: Agents to control ghosts
- graphicsDisplay.py Graphics for Pacman
- graphicsUtils.py: Support for Pacman graphics
- keyboardAgents.py: Keyboard interfaces to control Pacman
- layout.py: Code for reading layout files and storing their contents
- multiAgents.py: Where all of your multi-agent search agents will reside.
- pacman.py: The main file that runs Pacman games. This file also describes a Pacman GameState type, which you will use extensively in this project
- pacmanDQN_Agents.py: Implementation of the Deep Q-Learning agent
- textDisplay.py: ASCII graphics for Pacman
- util.py: Useful data structures for implementing search algorithms.

3.2 Neural Network Architecture

The Deep Q-Learning agent uses a neural network made originally by Tejas Kulkarni[27] and lately modified by Tycho van der Ouderaa[40]. In this section is explained in detail each layer of it.

First it is necessary to emphasise that this neural network is not a common one. Usually the neural networks are using raw pixel values from images[32] as an input but it is not the case now. Instead of the images the network is fed with a pool of matrix with the state of the map. As we can see in the figure 6 this matrix are fulfilled with 0s and 1s representing if there is something or not. Although we cannot notice about this, when we are playing our brain is probably

creating matrix like these. So actually is a really good representation of the state of the game. However, using this kind of matrix actually we are giving clues to the neural network about the game. This is the reason why usually neural networks are fed with images instead of matrix with 0s and 1s. Training with you can ensure a more realistic learning.

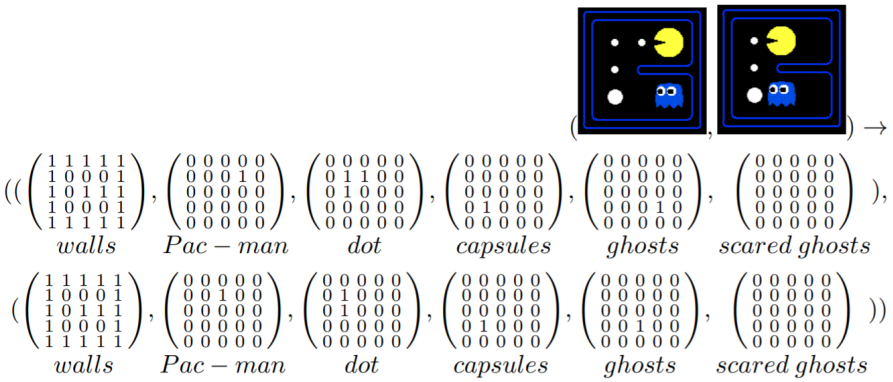

Figure 6: Pacman state representation

The Q-network consists of two convolutional layers followed by two fully connected layers. The convolutional layers are valuable because the game states in the Pac-man game imply the existence of local-connectivity. As a result, CNNs can be extremely useful to improve the classification of features in the game grid. The first layers use the Rectified Linear Unit (ReLu) non-linearity as activation function. The ReLu is defined as f(x) = max(0, x). The biggest advantage of ReLu is indeed non-saturation of its gradient, which greatly accelerates the convergence of stochastic gradient descent compared to the sigmoid / tanh functions[26]. The final layer is fully-connected and uses a linear activation function.

Layer	**Function**	**Input**	**Output**	**Filter size**
Conv	ReLu	W x H x 6	W - 2 x H -2 x 16	3 x 3
Conv	ReLu	W - 2 x H - 2 x 16	W - 4 x H - 4 x 32	3 x 3
Fully	ReLu	W - 4 x H - 4 x 32	256	
Fully	Linear	256	4	

Table 1: Neural Network Architecture

Finally, RMSprop[15] is used as an optimiser of gradient descent. In RMSprop gradients are divided by the mean squared error of the widths, in order to train more efficiently.

3.3 Pacman Parameters

The original Pacman environment give us the possibility to change some of the parameters in the game, using the package *argparse*, just putting them in the command when a Pacman instance is executed. To prepare the environment for the DGGA some parameters have been added in the parameters list. The following parameters are considered the most important parameters:

- --explore_action *arg*: Using this parameter you can choose which agent or agents you want to use in the exploration state of the algorithm. Default value is reflex. The options are random, reflex, minimax, random_reflex, reflex_minimax and random_reflex_minimax.
- --discount *arg*: This parameter sets the *gamma* (γ) probability in the reinforcement learning algorithm. The default value is 0.95. The range of this parameter is from 0 to 1.
- --lr *arg*: This parameter set the *learning rate* in the RMSporp optimizer for the neutal network. The default value is 0.0002. The range of this parameter is from 0 to 1.
- --rms_decay *arg*: This parameter is used by the RMSprop to make the learning rate decrease over each update. The default value is 0.99. The range of this parameter is from 0 to 1.
- --rms_eps *arg*: This parameter is used by the RMSprop, it is just a fuzz factor as described in the official documentation[39]. The default value is 0.000001. The range of this parameter is from 0 to 1.
- --eps *arg*: This variable corresponds to ϵ in the reinforcement learning part of the algorithm. With this variable is decided whether do exploration or exploitation. The default value is 1.0. The range of this parameter is from 0 to 1.
- --eps_final *arg*: This parameter reference to the lowest value for the variable Epsilon and must be higher than the parameter *eps*. The default value is 0.1. The range of this parameter is from 0 to 1.
- --eps_step *arg*: This parameter let you set how much ϵ is decreased over each iteration. The default value is 100000. The range of this parameter is from 0 to ∞.
- --level *arg*: This parameter lets you choose the agent execution level for the execution. The default value is 1. The options are 1, 2 and 3.

After doing some test we realised we needed to implement a way where we can easily tell the program when change between the exploration agents. It is important to take into account that the algorithm changes agents only when it is using the parameter *explore_action* with multiple agents for instance

random_reflex. Therefore, we create a simple scale of levels that you can select using the parameter *--level* when you execute an instance. These are the three levels implemented:

1. The level 1 is the deepest level. Each time the agent needs to get an action from the exploration state, randomly selects which exploration agent will handle this responsibility.

2. In the level 2 the algorithm changes the exploration agent every 50 games. So during 50 games the exploration agent will be the same.

3. When you select level 3 the algorithm select the exploration agent at the beginning. Then it is used for the full experiment.

Once the level is selected the algorithm is able to change the exploration agent. To change the exploration agent is used an ϵ_2 that is fixed in the algorithm. The ϵ_2 is used in the same way as RL uses ϵ. A practical case of this could be using the parameter *explore_action* with the value random_reflex. Firstly the ϵ decides between exploration and exploitation. If it decides for exploration then ϵ_2 is who decides to use either the random or the reflex agent. For example, putting the value of ϵ_2 to 0.3 we tell the algorithm to use the 30% of the times the random agent and the remaining 70% for the reflex agent.

In the next snipped we can see and example of a method to select the exploration agent. In this case, and according with the given example, we selected as example method *get_action_random_reflex()*.

```
def get_action_random_reflex(self, gameState):
    if np.random.rand() < 0.3:  # This epsilon 2
        return self.get_action_random(gameState)
    else:
        return self.get_action_reflex(gameState)
```

3.4 Applying DGGA

This section contains a simple explanation of how to execute *DGGA*. First of all, we discuss the execution parameters of DGGA. Then, we present the configuration files needed to describe the configuration or execution parameters of the target algorithm and the set of instances used to evaluate the performance of a particular genome or configuration.

Finally, we describe both how to adapt *DGGA* and the target algorithm on a particular distributed execution environment. This task is in charge of what we call execution environment wrappers.

3.4.1 DGGA execution parameters

DGGA inherits all the execution parameters of *GGA* extended with six new parameters. In what follows, we will show how to execute *DGGA*, and the list

of parameters with a brief explanation of each one.

```
./dgga <param_tree_file> <instances_seed_file> [parameters]
```

The first two parameters are mandatory and must be specified in the given order:

- `<param_tree_file>`: the path to the XML configuration file defining the parameter tree. See next section for a detailed explanation.
- `<instances_seed_file>`: the path to the instances seed file with the instances to tune the algorithm. See an example in next section.

The rest of the parameters are optional:

- `-g/--generations` *arg*: maximum number of generations (default value: 100).
- `-p/--population_size` *arg*: size of the initial population. This can vary over the course of the algorithm, but generally doesn't stray more than 25% (default value: 100).
- `--use-epigenetics` *arg*: If enabled the target algorithm environment will be extended with two new variables, GGA_EPI_IN and GGA_EPI_OUT, which are files that can be used to transfer information between different evaluations of the same genome and to its offspring (default: no)
- `-t/--num_threads` *arg*: mini-tournament size. Number of members to run simultaneously (default value: 4).
- `-w/--pct_winners` *arg*: percentage of winners per mini-tournament [0.0, 1.0] (default value: 0.125).
- `--is` *arg*: number of instances at the start of the tuning (default value: 5).
- `--ie` *arg*: number of instances at the end of the tuning (default value: 100).
- `--gf` *arg*: generation at which to reach the amount of instances specified with the parameter `--ie` (default value: -1).
- `--seed` *arg*: seed for the internal random engine (default value: current time in seconds).
- `--seeded_genomes` *arg*: number of "seeded genomes" to create from some set of default parameters (default value: 0).
- `--pe` *arg*: penalty applied to the evaluation of those genomes that reach the timeout time per instance (default value: 1.0).

- `--max_evals` *arg*: maximum number of evaluations allowed to find the best configuration. This is not strict, but a best effort will be made to come close to this number (default value: 2147483647).
- `-m/--mutation_rate` *arg*: the probability that a parameter is mutated when generating new individuals (default value: 0.1).
- `--st` *arg*: sub-tree split probability [0.0, 1.0] (default value: 0.1).
- `--sp` *arg*: sigma percentage. Determines sigma for the Gaussian distribution of the random engine (default value: 1.0).
- `-v/--verbosity` *arg*: verbosity level [0, 5] (default value: 2).
- `--ga` *arg*: maximum age of a genome (default value: 3).
- `--rt` *arg*: if false, *DGGA* tunes for output, rather than runtime. The last line of the algorithm output must be the objective value to minimize [true, false] (default value: true).
- `--nc` *arg*: if true, *DGGA* normalizes all the continuous variables (default value: false).
- `--su1` *arg*: if true, sends SIGUSR1 before sending SIGTERM to kill the evaluation processes (default value: false).
- `--ls` *arg*: specifies the learning strategy (default value: 1). {0 = TESTING, 1 = Linear, 2 = Step, 3 = Parabola, 4 = Exponential}
- `--lsd` *arg*: generation at which start the learning strategy. Until then `--is` instances will be used (default value: 0).
- `--lss` *arg*: number of generations per step for the step strategy (default value: 5).
- `--tacl` *arg*: target algorithm CPU timeout in seconds (default value: 30).
- `--tc` *arg*: tuner CPU timeout in seconds (default value: 2147483647).
- `--twc` *arg*: tuner wall-clock timeout in seconds (default value: 2147483647).
- `--conf_file` *arg*: specifies a configuration file. Command line options are preferred (default value: "").
- `--traj_file` *arg*: path of the trajectory file (default value: "").
- `--scen_file` *arg*: specifies a scenario file as used by ParamILS/SMAC. Note: this file overrides certain command line options (default value: "").
- `--master`: the presence of this flag indicates that the process must act as the Master of *DGGA*.

- `--worker`: the presence of this flag indicates that the process must act as a Worker of *DGGA*.
- `--ip` *arg*: specifies an IP to connect to the Master. It can be used several times to specify more than one IP.
- `--port` *arg*: specifies the port to connect to the Master (default value: 6789).
- `--nodes` *arg*: specifies the desired number Workers. It is mandatory when "--master" is specified.
- `--start-worker-wrapper` *arg*: specifies the path to the wrapper that the Master will use to start the Workers (default value: "").
- `-h/--help`: prints the help message and exits.

3.4.2 Tuning configuration files

DGGA depends on two main configuration files. A XML configuration file, with the configuration of the target algorithm, and an instances seed file, with the list of instances to tune the algorithm.

3.4.2.1 XML configuration file This file provides to *DGGA* all the necessary information about the target algorithm and its configuration or execution parameters. It also provides user defined parametrizations and the command to run the algorithm. The format of this file, as mentioned before, is XML

Below, we show and describe the structure of the XML file:

```
<?xml version="1.0" encoding="UTF-8"?>
<algconf>
   <cmd executable="python3">
      <arg value="${instance}" />
      <arg value="-q" />
      <arg value="--load_file model_2k" />
      <arg value="${train_start}" prefix="--train_start " />
      <arg value="${batch_size}" prefix="--batch_size " />
      <arg value="${mem_size}" prefix="--mem_size " />
      <arg value="${discount}" prefix="--discount " />
      <arg value="${lr}" prefix="--lr " />
      <arg value="${rms_decay}" prefix="--rms_decay " />
      <arg value="${rms_eps}" prefix="--rms_eps " />
      <arg value="${eps}" prefix="--eps " />
      <arg value="${eps_final}" prefix="--eps_final " />
      <arg value="${eps_step}" prefix="--eps_step " />
      <arg value="${explore_action}" prefix="--explore_action " />
      <arg value="${minimax_depth}" prefix="--minimax_depth " />
   </cmd>
   <seedgenome default="false">
      <param name="root" value="0" />
      <param name="train_start" value="10000" />
      <param name="batch_size" value="32" />
      <param name="mem_size" value="100000" />
```

```
      <param name="discount" value="0.95" />
      <param name="lr" value="0.0002" />
      <param name="rms_decay" value="0.99" />
      <param name="rms_eps" value="0.000001" />
      <param name="eps" value="1.0" />
      <param name="eps_final" value="0.1" />
      <param name="eps_step" value="100000" />
      <param name="explore_action" value="random" />
      <param name="minimax_depth" value="1" />
   </seedgenome>
   <ptnode type="and" name="root" domain="[0, 0]">
      <ptnode type="and" name="train_start" domain="[5000,15000]" />
      <ptnode type="and" name="batch_size" domain="[16,48]" />
      <ptnode type="and" name="mem_size" domain="[50000,150000]"  />
      <ptnode type="and" name="discount" domain="[0.0, 1.0]" />
      <ptnode type="and" name="lr" domain="[0.0001, 0.001]" />
      <ptnode type="and" name="rms_decay" domain="[0.99,0.99]" />
      <ptnode type="and" name="rms_eps" domain="[0.0, 1.0]" />
      <ptnode type="and" name="eps" domain="[1.0, 1.0]" />
      <ptnode type="and" name="eps_final" domain="[0.00001, 1.0]" />
      <ptnode type="and" name="eps_step" domain="[50000,150000]" />
      <ptnode type="or" name="explore_action" domain="{random,reflex,minimax}">
            <ptnode type="and" name="minimax_depth" domain="[1,2]" or-domain="{minimax}"/>
     </ptnode>
   </ptnode>
</algconf>
```

All the specification of the target binary is enclosed within the $< algconf >< /algconf >$ tags. The second pair of tags of the file, $< cmd >< /cmd >$, contain the command that DGGA must use to execute the target algorithm. The variables *$instance* and *$seed* are special variables that tell DGGA where to put the path to the instance and the seed. The rest of the parameters, for example train_start, correspond to parameters to be tuned. Prefix indicates what text to prefix to the parameter in the command line.

Next, the $< seedgenome >< /seedgenome >$ tag specifies settings of the parameters to insert into the initial population. Each $< param/ >$ tag, specifies the name of a parameter and the value for it to take in the genome.

The next portion of the file specifies the parameter tree itself. The tree is specified by a single node corresponding to the root of the tree. All node tags that are not a child of this root node will be ignored. Each $< ptnode >$ tag my have any number of nodes underneath it. $< ptnode >$s may have one of two types: "and" or "or", which will be described in a moment. Each node is either categorical, discrete or continuous. Categorical nodes have the *"categories"* attribute specified (see *explore_action*). Discrete parameters have "start" and "end" specified with integer values (see *batch_size*). Floating point parameters are specified like discrete parameters, except with floating point numbers in start and end (see *rms_eps*).

As mentioned before a node can be either an "and" node or an "or" node. An "and" node indicates that the child nodes should all be present whenever the parent node is present in the parameter settings of a genome. An "or" node means that only the node corresponding to the selected branch should be included

in the settings. In other words, or nodes allow users to select between categorical parameters, and associate a branch of the parameter tree with the parameter. This relationship is used within the optimization to find good parameters. In the example above, *explore_action* has two settings: 'random' ,'reflex' and 'minimax'. Setting 'minimax' is associated with the branch of *minimax_depth*.

3.4.2.2 Instances seed file This file contains lines with pairs of values, the first element is the seed of the instance and the second is the path to the instance file. These values correspond to the variables *$seed* and *$instance* mentioned in the previous point.

Below, we show the required line for the instances files:

```
0 /home/student/Merino/PacmanDQN/pacman.py
```

3.4.3 Execution environment

DGGA can not modify the behaviour of the target algorithm and does not know how to start a Worker in an specific architecture. For these reasons the user has to create wrappers to adapt *DGGA* and the target algorithm.

The language used to create the wrappers does not matter as long as it is executable like a native binary. Since the wrappers do not have to perform resource consuming tasks, we recommend to use a scripting language such as: Bash [9] or Python [41].

As we lauched the Pacman project directly from the Worker, it has not needed follow the usual way and we do not create a wrapper.

This version of DGGA have an important variation from the last version of DGGA as it integrates the parameter *use-epigenetics* and we use it. As it was explained in the section 3.4.1, the target algorithm environment will be extended with two new variables, GGA_EPI_IN and GGA_EPI_OUT, which are files that can be used to transfer information between different evaluations of the same genome and to its offspring.

To use DGGA with the parameter *use-epigenetics* is required to include in the algorithm methods to get the variables GGA_EPI_IN and GGA_EPI_OUT from the Linux environment, in this case it has been used the variable *environ* from the python package *os*.These variables are taken at the beginning of the execution, when we extract all the information from the Pacman parameters. These variables are giving the names for the input and output neural network model names respectively. the The GGA_EPI_IN variable has the path of model that will be used as a pre-trained model in the execution, that means the execution starts in an advanced state. The GGA_EPI_OUT holds the name for the model that will be stored at the end of the execution. A part from these variables, we take also the variable *GGA_INDIVIDUAL_ID* from the Linux environment. This variable, added by DGGA, give us the id of the current execution and it is used to sort the logs and results generated files names. In the case of any of these variables are not in the variable *environ*, the algorithm do not load/save any model and just execute the instance.

Below, we show an example how to get these variables using the variable *environ*:

```
meta_load = os.environ["GGA_EPI_IN"] if "GGA_EPI_IN" in os.environ else ""
meta_save = os.environ['GGA_EPI_OUT'] if "GGA_EPI_OUT" in os.environ else ""
individualID =
    os.environ['GGA_INDIVIDUAL_ID'] if "GGA_INDIVIDUAL_ID" in os.environ else 'None'
```

4 Experiments and Results

This section describes the extensive experimentation performed during the project. We first describe some basic concepts that we will use for the sake of clarity. Then, we introduce the set of experiments, describing the goal, the settings and the result obtained.

We run our experiments in a computer cluster with each node equipped with two octo-core Intel Xeon Silver 4110 @ 2.10 GHz processors and 96 GiB of RAM. It runs a copy of RocksCluster 7.0 and uses SGE to manage jobs.

As we described in the section 3.1, we have the folder called *layouts* where there are all the available layout files to use in the experiments. We focus our experiments with variation of the map called *mediumClassic*. This map has 11 rows and 20 columns. Originally it has 2 ghosts but we are added another ghost to do the experiments more challenging.

4.1 Basic definitions

A *game* or episode consist on the classic game played in Pacman where the agent navigates Pac-Man through a maze containing various dots, known as Pac-Dots, and ghosts. The goal of the game is to accumulate points by eating all the Pac-Dots in the maze. However, the ghosts roam the maze, trying to kill Pac-Man. If any of the ghosts hit Pac-Man the game is over.

An scenario $\langle \{f_1, \ldots, f_v\}, N' \rangle = SC(u,v,N)$ involves the execution of the DQL agent (Pacman agent) on u training games (or episodes) followed by its execution on v test games. During an initial training phase, the DQL agent learns how to efficiently play the Pacman game, using both exploration and exploitation in the part of action decision, and during the test phase, where the algorithm only uses the neural network to decide which is the best action for the next movement, we evaluate how good that learning was.

In order to report the accuracy on the test games, from each scenario we store the remaining food after the game is over, being f_i the amount of food in the map when Pacman either wins, loses or time is over. In the original Pacman environment there is a parameter to assign a timeout to the agent to take an action, by default is 30 seconds. By using a similar technique in the class *game.py* we have implemented a timeout for the entirely game not just for the agent. Doing this we are ensuring the experiments will not last for ever. The default value for this timeout is 60 seconds. As memory limit we used 4GB to run each game.

The input N is the set of weights used to initialise the neural network. The output N' is the set of weights of the neural network after playing the u training games.

4.2 Getting points for the plots

The values to create all the charts were extracted doing the average every 50 games. All the experiments have done a number of games multiple of 50 to

ensure no avoid any played game in the results.

One of the main points of this technique is that the games are done in test mode, which means that the ϵ is set to 0 to make the algorithm do always exploitation. Doing that we can make sure that the results come from the neural network, which are we testing, and not from the exploration agent.

4.3 Experiment 1 - Random Sampling

Goal: To confirm our hypothesis that different parameterizations of the DQL algorithm can dramatically impact on the DQL agent's performance.

Before getting familiar with and using a more powerful AC tool, like DGGA, we randomly generate some configurations to confirm if there is indeed potential in the search space of the possible parameterizations for the DQL algorithm.

Setting: We will experiment with scenarios of 2000 training games and 200 test games. We set the timeout of each game ∞. Then, taking the default parameterization as our base case, we generate several parameterizations by mutating randomly just one single hyperparameter.

Results: In figure 7, we present the results of our experiment. In the x-axis we present the default parameterization, and for each of the randomly generated new parameterizations we show the value of the mutated hyperparameter. Since the timeout is ∞, we show in the y-axis the average number of wins on the test games we achieved on the scenario with a given parameterization. In order to compute the average number of wins, we just count the number of maps with food equal to 0 when the game ends.

As we can observe in the figure, changing the hyperparameter *eps* the agent is obtaining a 20% more win rate than using the default configuration. In contrast, when we look at the hyperparameter *eps_steps* the results are not so impressive, as the win rate is a 0% that means the agent was not be able to win any games.

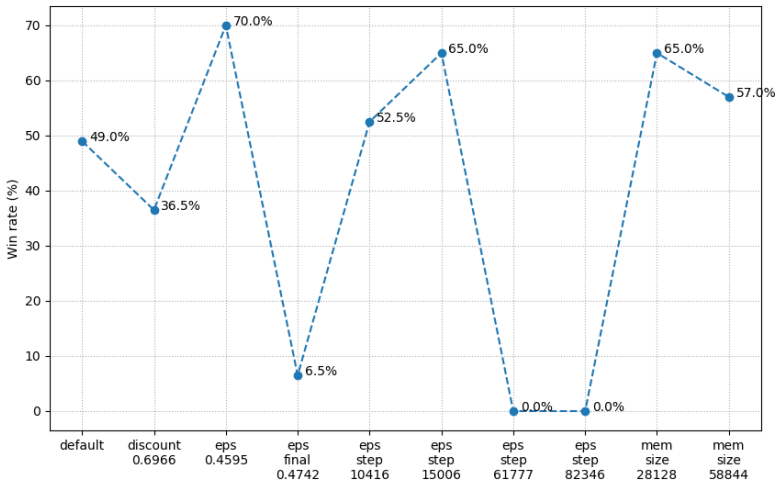

Figure 7: Win rate for randomly sampled parameterizations of the DQL agent.

This experiment demonstrates that indeed different parameterizations can have a dramatic impact. It comes clear to us, that, just with subtle changes of the hyper-parameters values the agent is able to acquire better results than using the default configuration.

4.4 Experiment 2 - Impact of alternative strategies for the exploration phase

Goal: To identify which strategy should we use for the exploration phase.

As we described previously, Q-Learning alternates exploration and exploitation phase. While for the exploitation phase the DQL agent will take actions accordingly to the learnt policy, for the exploration phase typically random actions are taken. However, we may use some other strategies or policies. In particular, we will also experiment with a Reflex and Minimax agents both described in the Berkeley project [25].

Setting: We will experiment with a sequence of 280 scenarios consisting each of 50 training games and 50 test games. Each scenario is initialised with the status of the network of the preceding one. For the first scenario, the weights are set randomly. Therefore, we will perform 14000 training and 14000 test games. The timeout for each game was set to ∞. We used the default parameterization of the DQL agent, except for hyper-parameter that defines which exploration strategy we will use (*explore_action*). We experiment with the values: random, minimax and reflex.

Results: In figure 8, we present the results of our experiment. In the x-axis we present the number of played training games. For each scenario, since the timeout if ∞, we can compute the average number of wins. Then, in the y-axis we present the maximum average number of wins seen so far.

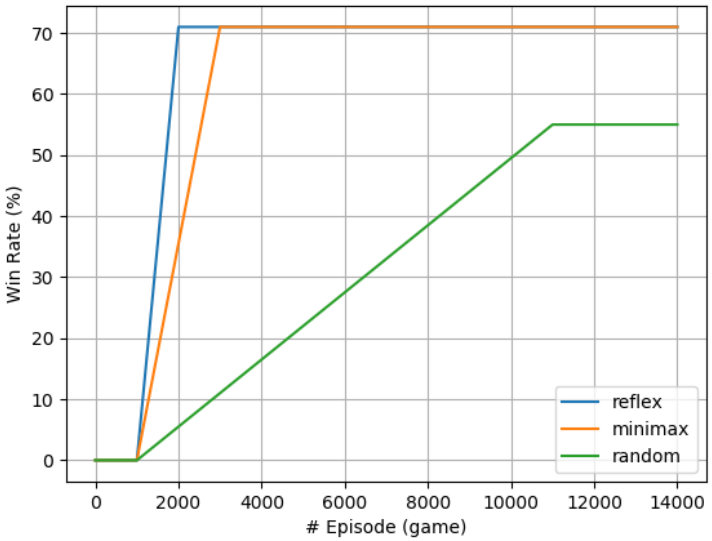

Figure 8: Win rate using random, minimax and reflex exploration.

As we can see, the DQL agent is unable to learn anything useful till at least 1000 training games have been played. Respect to the effectiveness of the different choices for the exploration strategy, it is clear that both reflex and minimax are superior to random. However, reflex needs less training games to achieve the maximum average number of wins, 71%.

In figure 9 we computed for each scenario the average remaining food and we plot the minimum average seen so far. As we can see, all of the agents start to learn that is useful for them eat food. We can see easily how linked are the average wins and the average remaining food, as the more win rate, the less food rate. Respect to the effectiveness of the different choices for the exploration strategy, it is clear that both reflex and minimax are superior to random as we also observed in figure 8. Regarding to the food rate, reflex and minimax achieve similar results.

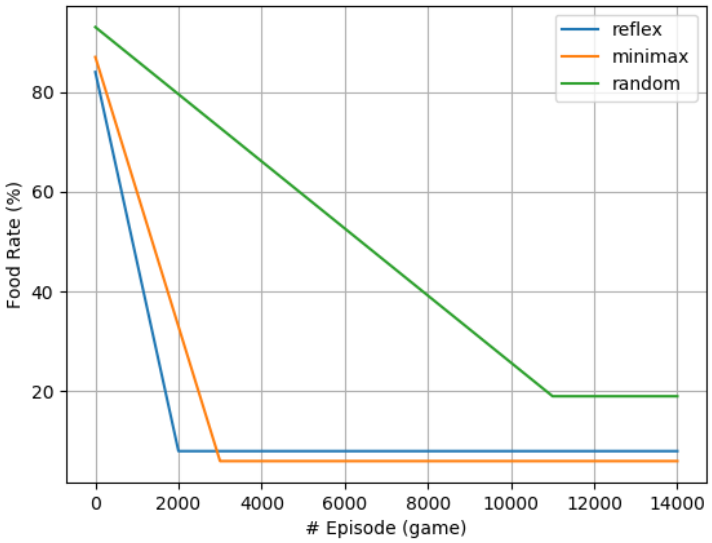

Figure 9: Food rate using random, minimax and reflex agent exploration.

Finally, in figure 10 we focus on the reflex strategy and show both the evolution of the average wins and average remaining food. As we can observe, the agent is not able to start winning until 1000 games. However, from the start it learns it should eat as much food as possible and in consequence, its average wins increases.

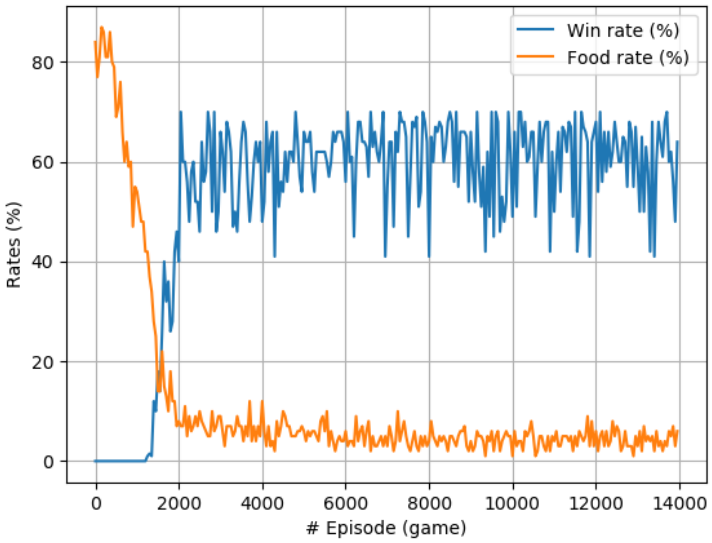

Figure 10: Evolution of the average wins and average remaining food on the reflex strategy.

4.5 Experiment 3 - Impact of Automatic Configuration

Goal: To asses the impact of automatically configuring the DQL agent for the Berkeley Pacman project.

Based on the results of the previous experiment, we will use as an starting point or seed genome for DGGA the status of the DQL agent with exploration strategy reflex after 2000 training played games.

Setting: We will experiment with scenarios of 100 training games and 100 test games. We set the timeout of each game 60 seconds. The model trained with 2000 games and reflex agent in the exploration phase is used to do this experiment. We fix the hyper-parameter *explore_action* to force DGGA to use either random, reflex or minimax as agent for the exploration phase and create the models *Random model*, *Reflex model* and *Minimax model*. The model generated using the parameter *explore_action* free, which can use every kind of exploration agent for the exploration phase, is called *DGGA model*.

Results: Observing the results, firstly we should say that fixing the hyper-parameter *explore_action* the results for the *Random model*, *Reflex model* and *Minimax model* are considered a fail as the agents were not able to increase its performance more than a 5%, even the *Random model* and *Minimax model* have decreased their performance. What is really impressive is the improvement of

the *DGGA model* with a 12% more win rate than the *base model*.

In the table 2 we can see the different values for the average wins and average remaining food for each experiment. We must say that the *DGGA model* has improved having now the win rate to 83% whilst the *Base agent* just has a 71%. If we observe the third column, we can notice *DGGA model* is not the best agent eating food, in this case is the experiment fixing the hyper-parameter *explore_action* to *reflex agent*.

Model name	**Win_rate (%)**	**Left_Food_rate (%)**	**Iterations**
Base model	71	4	2000
DGGA model	**83**	3.3	4400
Random model	69	5.04	2400
Reflex model	73	**2.8**	4200
Minimax model	68	3.3	3900

Table 2: Results fixing the exploration agent using DGGA

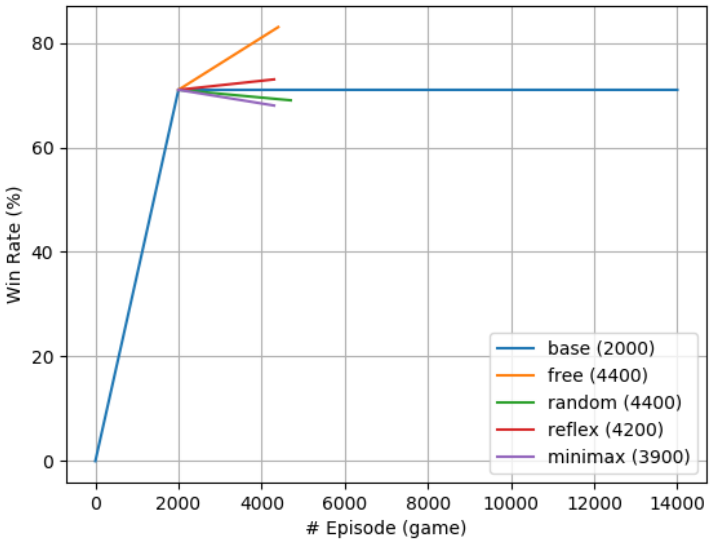

Figure 11: Results fixing the exploration agent using DGGA

Observing the figure 11 we can notice how better is the *DGGA model* than the *Base model*. What is more, the *DGGA model* has played 10000 games less than the other. In this case we see how useful useful is DGGA in this experiment for two reasons. The first one and the most important, the win rate is much better than using just the default configuration. The second reason is when we look the total execution time. To obtain the *Base model* we needed many more hours than the *DGGA model* as it has much less played games.

It is important to say that the models trained with DGGA have much less number of played episodes due to DGGA has a threshold to determine whether the model it is training could learn more or instead, it has obtained its best performance and then DGGA finishes its execution giving the name of the best model.

5 Conclusions and future work

Through the development of this project, we have seen that automatic configuration is crucial to exploit all the capabilities of a Deep Q-Learning agent.

Thanks to the first experiment we were able to determine that a DQL agent can achieve different better performance just randomly muting some hyper-parameter from the default configuration. We found a configuration to confirm there is indeed potential in the search space of the possible parameterizations for the DQL algorithm.

By doing the second experiment we can notice the impact of alternative strategies for the exploration phase. We found that for the Berkeley Pacman project is better use the *reflex* agent instead of the *random* agent in the exploration phase of the DQL agent.

After the third and last experiment we observed that DGGA was able to seek an optimal configuration for the learning hyper-parameter obtaining a considerable improvement in respecting of using the default configuration. We have seen it is not needed to create an agent with many games, if you can make one with less games and then apply DGGA to it to improve its performance. We could appreciate this with the *DGGA model* in section 4.5.

As future work, we have several working avenues, which is a sign of the potential of this project:

- Finish the Pytorch implementation we started and compare which platform is better either Tensorflow or in reality there is no difference.

- Now at the end of every DGGA scenario, we are telling DGGA what is the average remaining food for the scenario after evaluate the final DQL model. One new approach can be modifying the information we give to DGGA at the end of the scenario to create a model evaluating both parameters, the average wins and the average remaining food.

- In this case we are using the parameter epigenetics for DGGA. This parameter adds to the Linux environment new variables as we explained in section 3.4.3. At the moment we are saving into the variable GGA_EPI_OUT the final neural network after finish the scenario. A new experiment that can be performed could be, after evaluate the final neural network, save in the GGA_EPI_OUT the best model either the initial or the final. Saving always the final we are not ensuring we are taking the best model as the initial neural network could become worse at the end of the scenario.

Finally, this project has shown me how it works a real research study. Also I found all that I have done within this work somehow interesting, but as any project, this one also had its pros and cons. Writing this report has been tedious and difficult because I am not used to write this kind of documents. However, my final impression is positive because I have been working in what I like and I think that the final result is a useful research project.

References

[1] B. Adenso-Diaz and M. Laguna. Fine-tuning of algorithms using fractional experimental design and local search. *Operations Research*, 54(1):99--114, 2006.

[2] Belarmino Adenso-Diaz and Manuel Laguna. Fine-tuning of algorithms using fractional experimental designs and local search. *Oper. Res.*, 54(1):99--114, January 2006.

[3] C. Ansotegui, M. Sellmann, and K. Tierney. A gender-based genetic algorithm for the automatic configuration of algorithms. In *Proceedings of the 15th International Conference on Principles and Practice of Constraint Programming*, pages 142--157, 2009.

[4] Carlos Ansótegui, Meinolf Sellmann, and Kevin Tierney. A gender-based genetic algorithm for the automatic configuration of algorithms. In *Proceedings of the 15th International Conference on Principles and Practice of Constraint Programming*, CP'09, pages 142--157, Berlin, Heidelberg, 2009. Springer-Verlag.

[5] M. Birattari, Z. Yuan, P. Balaprakash, and T. Stützle. F-race and iterated f-race: An overview. In *Empirical Methods for the Analysis of Optimization Algorithms*, pages 311--336, 2010.

[6] Mauro Birattari, Thomas Stützle, Luis Paquete, and Klaus Varrentrapp. A racing algorithm for configuring metaheuristics. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '02, pages 11--18, San Francisco, CA, USA, 2002. Morgan Kaufmann Publishers Inc.

[7] Steven P. Coy, Bruce L. Golden, George C. Runger, and Edward A. Wasil. Using experimental design to find effective parameter settings for heuristics. *Journal of Heuristics*, 7(1):77--97, January 2001.

[8] T. Fitzgerald, Y. Malitsky, B. O'Sullivan, and K. Tierney. React: Real-time algorithm configuration through tournaments. In *Proceedings of the Symposium on Combinatorial Search*, 2014.

[9] Brian Fox. Bash (unix shell). `http://www.gnu.org/software/bash/`, 1989-2014.

[10] Alex S. Fukunaga. Automated discovery of local search heuristics for satisfiability testing. *Evol. Comput.*, 16(1):31--61, March 2008.

[11] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier neural networks. In Geoffrey Gordon, David Dunson, and Miroslav Dudík, editors, *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, volume 15 of *Proceedings of Machine Learning Research*, pages 315--323, Fort Lauderdale, FL, USA, 11--13 Apr 2011. PMLR.

[12] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1989.

[13] Carla P. Gomes and Bart Selman. Algorithm portfolio design: Theory vs. practice. *CoRR*, abs/1302.1541, 2013.

[14] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770--778, June 2016.

[15] Geoffrey Hinton. RMSpromp original slides. `https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf`, 2012.

[16] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735--1780, 1997.

[17] G. Huang, Z. Liu, L. v. d. Maaten, and K. Q. Weinberger. Densely connected convolutional networks. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2261--2269, July 2017.

[18] Bernardo A Huberman, Rajan M Lukose, and Tad Hogg. An economics approach to hard computational problems. *Science*, 275(5296):51--54, 1997.

[19] F. Hutter, H.H. Hoos, and K. Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *Learning and Intelligent Optimization - 5th International Conference, LION 5, Rome, Italy, January 17-21, 2011. Selected Papers*, pages 507--523, 2011.

[20] F. Hutter, H.H. Hoos, K. Leyton-Brown, and T. Stuetzle. ParamILS: An automatic algorithm configuration framework. *JAIR*, 36:267--306, 2009.

[21] Frank Hutter, Holger H. Hoos, and Thomas Stützle. Automatic algorithm configuration based on local search. In *Proceedings of the 22Nd National Conference on Artificial Intelligence - Volume 2*, AAAI'07, pages 1152--1157. AAAI Press, 2007.

[22] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *Proceedings of the 32Nd International Conference on International Conference on Machine Learning - Volume 37*, ICML'15, pages 448--456. JMLR.org, 2015.

[23] K. Jarrett, K. Kavukcuoglu, M. Ranzato, and Y. LeCun. What is the best multi-stage architecture for object recognition? In *2009 IEEE 12th International Conference on Computer Vision*, pages 2146--2153, Sept 2009.

[24] Pieter Abbeel John DeNero, Dan Klein. Berkeley Pacman Project website. `http://ai.berkeley.edu/project_overview.html`, 2014.

[25] Pieter Abbeel John DeNero, Dan Klein. Project 2: Multi-Agent Search website. `http://ai.berkeley.edu/multiagent.html`, 2014.

[26] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097--1105. Curran Associates, Inc., 2012.

[27] Tejas Kulkarni. DQN class. `https://github.com/mrkulk/deepQN_tensorflow/blob/master/DQN.py`, 2016.

[28] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521:436 EP --, May 2015.

[29] Radu Marinescu and Rina Dechter. And/or branch-and-bound search for combinatorial optimization in graphical models. *Artificial Intelligence*, 173(16–17):1457 -- 1491, 2009.

[30] Warren S. McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115--133, Dec 1943.

[31] Steven Minton. Automatically configuring constraint satisfaction programs: A case study. *Constraints*, 1(1-2):7--43, 1996.

[32] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013.

[33] A. Ng. Improving deep neural networks: Hyperparameter tuning, regularization and optimization. `https://www.coursera.org/learn/deep-neural-network`, 2018. Coursera.

[34] Mihai Oltean. Evolving evolutionary algorithms using linear genetic programming. *Evol. Comput.*, 13(3):387--410, September 2005.

[35] Mike Preuss and Thomas Bartz-Beielstein. Sequential parameter optimization applied to self-adaptation for binary-coded evolutionary algorithms. In Fernando G. Lobo, Cláudio F. Lima, and Zbigniew Michalewicz, editors, *Parameter Setting in Evolutionary Algorithms*, volume 54 of *Studies in Computational Intelligence*, pages 91--119. Springer, 2007.

[36] F. Rossi, P. van Beek, and T. Walsh, editors. *Handbook of Constraint Programming*. Elsevier, 2006.

[37] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211--252, 2015.

[38] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In *ICLR*, 2015.

[39] TensorFlow. TensorFlow Documentation. https://www.tensorflow.org/api_docs/python/tf/train/RMSPropOptimizer, 2018.

[40] Tycho van der Ouderaa. Deep reinforcement learning in pac-man. *None*, 2016.

[41] Guido van Rossum. Python programming language. https://www.python.org/, 1991-2014.